

# True XML Web Services

*Keywords:* Distributed Systems, SOAP, Web Services, WSDL, XML-RPC, DocSOAP, XDK, Manifest, Document Framework, stream processing, Castor, JAXB, XGen

Michael Leventhal

U.S.

[michael@textscience.com](mailto:michael@textscience.com)

<http://www.textscience.com>

## *Biography*

Michael Leventhal led the creation of Commerce One's open source DocSOAP XDK - a SOAP, XML, and Web Services high-performance, document-centric toolkit. Leventhal was also deeply involved with the Mozilla open source project and has architected and led numerous projects in the area of Web applications and infrastructure and XML (and SGML) over the last ten years, developed and taught the first university-level course in XML and wrote the first book on XML software development for the Internet . He has given many presentations at previous XML conferences and other conferences and user groups. Leventhal holds a degree in Electrical Engineering and Computer Science from U.C. Berkeley.

Sen Zhang

Dante Consulting, Arlington, Virginia

U.S.

[senzhang@yahoo.com](mailto:senzhang@yahoo.com)

<http://senzhang.netfirms.com>

## *Biography*

Sen Zhang is currently working as an Senior Principal for Dante Consulting. He architected Commerce One's open source DocSOAP XDK - a SOAP, XML, and Web Services high performance, document-centric toolkit - and led its implementation. His experience over the past ten years includes XML based Web messaging, E-Commerce applications, Computer Visualization and Knowledge Based Systems. Zhang received his MS in Computer Science for work on distributed computing systems from Iowa State University.

---

## Abstract

---

Document-style Web Services support document-centric applications - applications which exchange and process documents described by a wide range of schemas and which are much larger and more complex

than a SOAP-RPC payload. Document-style Web Services prove to be the best solution for a wide range of usage scenarios in b2b and may be more suitable for applications which do not easily reduce to service invocations at the granularity of RPC or are beset by some the performance challenges largely implicit in the remote procedure model.

Several techniques are presented for implementing an efficient and reliable document-centric Web Services framework: use of a manifest for ensuring message integrity and optimizing message processing; efficient parsing strategies for SOAP messages with large document payloads and/or attachments; use of a uniform document framework unifying document representations and supporting "lazy evaluation"; a comprehensive model for intramessage references; and the use of generated XML schema APIs to enable rapid programming of document-centric Web Services applications.

An open-source Java toolkit that the authors have developed, the DocSOAP XDK is presented. The DocSOAP XDK's implementation of many of the critical features needed in a high-performance, reliable SOAP framework is described. Finally, the authors propose a list of action items for fostering the widespread use of document-centric Web Services.

---

## **When a Protocol has the Great Virtue of being Worse than all its Competitors**

### **The Many-Colored Rainment of Web Services**

What are Web Services? The question is still troublesome as the answers are substantially and substantively different whether we consider the Web Services of specifications, of implementations, of marketing propaganda, or of the expectations of prospective users. In this paper we are going to develop a conception of Web Services which is different from the one which is most dominant today but which is solidly grounded, if not completely described, in the specifications, and best addresses the needs and expectations of users.

Before we can get there, however, we need to understand the dominant paradigm of today and why it can't deliver on the promise of Web Services.

### **Veni, Vidi, I SOAPed**

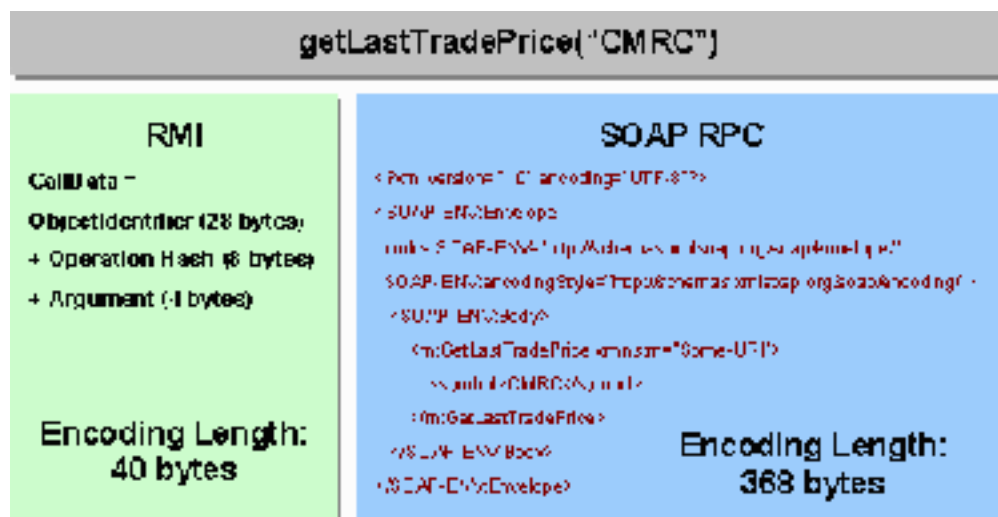
UDDI: service discovery, WSDL: service description, SOAP: service invocation - the three pillars of open-system standard service provision across the Internet. SOAP - Simple Object Access Protocol: services are provided by *objects* and accessing them will be *simple*. There are already many ways to access and invoke objects across networks: RMI, CORBA, RPC, DCOM. SOAP, however, tarts up remote procedure calls with XML; lovely to some but ineffective. More than anything else, it puts the *Simple* into SOAP. XML lends its virtues: it is self-describing, self-contained, transport-independent, reader-friendly, uses the existing web infrastructure, may use existing transport stacks, and passes through firewalls. A few crusty malcontents grumble that they don't see any difference between the old post/cgi mechanism and the new golden boy but most see SOAP as a big improvement. Anyone capable of sending a web page can invoke a

Web Service in a blink of the eye. A fine new toy appears on the web - but is it robust enough to be the cornerstone of the next generation of internet commerce and communication?

## A Fundamental Mismatch

We think that using XML to encode RPC is a misuse (or perhaps *non-use*) of the strengths of XML technology and, most of the time, simply not the solution to the problems posed by the next generation of internet applications. Here are five reasons for our position:

1. *Well, SOAP-RPC is slow.* Although advocates for SOAP-RPC began by admitting that it would be slow, given the inherent verbosity of XML, now that we know how slow it actually is no one can stand it. Quite a few studies have been done, see, for example, Davis and Parashar [\[CAIP\]](#) who demonstrate SOAP-RPC object invocations taking for 10 to 100 times as long to process as JavaRMI or CORBA. The figure below illustrates, for the usual example of a stock price quote, the differences in encoding style and size between RMI and SOAP-RPC.



2. *SOAP-RPC promotes a fine-grained approach to distributed computing over Wide Area Network (WAN).* Designing operations at the most efficient level of granularity is a challenging task for experienced computer programmers. In general, in order to make method invocations comprehensible to programmers the number of input arguments they carry and the complexity of the output is quite limited. Methods are fine-grained, that is, they perform small chunks of processing on very constrained input parameters. Many fine-grained transactions across a network are both extremely wasteful of network resources and also extremely unsatisfactory to the user who has to wait an inordinate amount of time to obtain a simple result. It may not even be possible in many cases to reduce a useful Web Service to single SOAP-RPC method invocation. Davis and Parashar [\[CAIP\]](#) found, in fact, that a significant cause for poor SOAP-RPC performance was that multiple calls were needed to effect one "logical" Web Services transaction.

3. *SOAP-RPC tightly couples the environment of service provider and service consumer.* While a major concept in Web Services is to decouple the service definition from the service implementation, in practice the RPC-style service definition tightly constrains the implementation. Since the WSDL service description binds both the provider and the consumer once a Web Service is deployed it will be nearly impossible to modify the method signature. Consumers of that Web Service program to it at the level of the exact method invocation. It is very difficult to handle the service invocation in a modular way that would limit the impact of any change. The technique of separating the interface from the implementation works well in closed and controlled environments but does not translate well into the dynamic and distributed world of Web Services.
4. *SOAP-RPC can severely impact performance as perceived by the user because of synchronous blocking calls.* Most computer programs run more or less sequentially; that is, an operation cannot start until the operation before it completes. While it is possible to use extremely sophisticated programming techniques to not suspend program execution while waiting for a SOAP-RPC method invocation to complete the vast majority of programs will simply perform a blocking call. Blocking call should be avoided at all cost when invoked over the Internet, which is both unreliable and unpredictable in latency .
5. *Translation between XML and programming lanaguage datatypes is not fullyinteroperable.* SOAP 1.1 embedded a set of encoding rules for mapping between SOAP-RPC XML and the simple types used in programming languages. An enormous amount of effort has been expended by members of SOAPBuilders in interoperability testing of RPC-encoded SOAP messages. Yes, the WS-I, a consortium of over 100 companies involved in Web Services, declared the SOAP 1.1 RPC-encoding rules impossible to implement in a full interoperable manner. While efforts to develop an interoperable mapping of simple types continues (see, for example, JAX-RPC) there is no agreed-upon, interoperable mapping for complex datatypes.

## Document-Style: The Other Web Service Paradigm

### Document-Style Web Services are Document-Centric

We introduce the idea of *document-centric* Web Services. Document-style specifies a few details of the message structure whereas the term document-centric describes the nature of the message construction, exchange and consumption which takes places between producer and consumer of services. Document-centric Web Services are not RPC; the payload does not contain a serialized method invocation. While SOAP is still an extremely useful message format it is no longer a *simple object access protocol*. What SOAP does provide includes the following:

- a network chain processing model
- a way to use existing transport stacks while remain independent of the specific transport
- a simple message type consisting of an XML document payload

- a way to associate message processing directives with the payload
- a way, through standards related to SOAP, to package attachments with the SOAP envelope

Document-centric Web Services are simply a message exchange paradigm where the exchange parameters are described by SOAP headers and the SOAP processing model and the message payloads are related schema-described XML documents and non-XML documents. The service request is described by the totality of the message - it may be far more complex than a function invocation. The service response, if any, is also described by the totality of the response message - it may be far more complex than the a value or values returned in a simple datatype by a function invocation.

## **A Better Match**

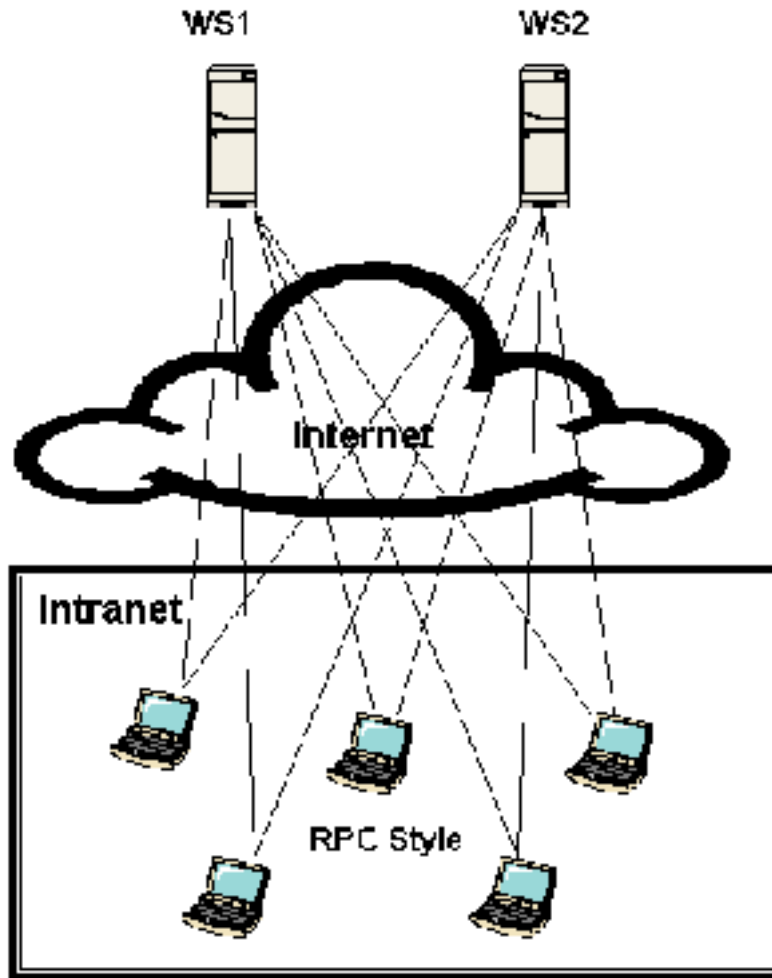
Document-centric Web Services, approached with some care, solve the problems associated with SOAP-RPC.

Document-centric Web Services should be coarse-grained. Commerce One's Web Services platform, as an example, exchanges business documents that are usually about 500 kilobytes. Processing this volume of information using fine-grained SOAP-RPC would require thousands of messages!

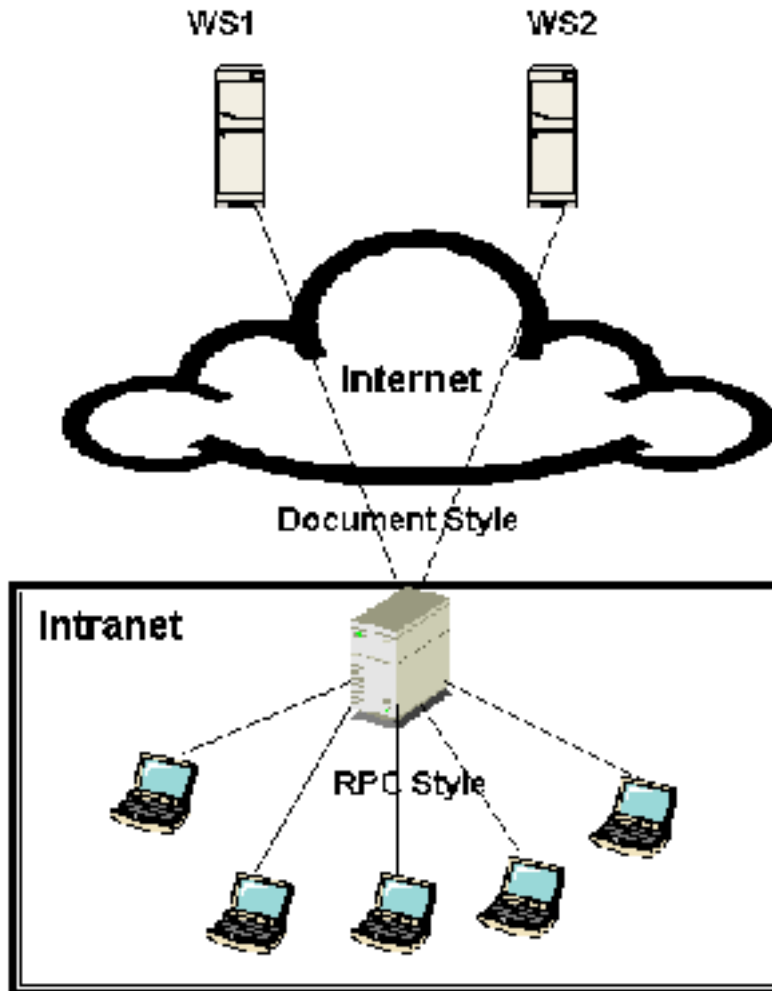
Burner [\[ACM\]](#) advises:

Because of the relatively high latency of access to web services, they should not be designed with "chatty" protocols. Do not design services to expose interfaces making small updates to data elements; rather, use generic "update" methods that can accept a transformed element in a single service call.

We illustrate the idea of aggregating information to reduce network traffic and request latency in the next two figures. The first shows a usual "chatty" protocol with a large number of users within an enterprise going out to the Internet to request fine-grained services. In this model the entire network will suffer degradation and each user will experience slow response times.



In the second image a large quantity of information is aggregated in a single request in document-centric Web Service message exchange between a process in the enterprise and the service providers. Enterprise members may use finer-grained RPC-Style Web Services to get information from the central server or other LAN-based methods to retrieve information.



The verbosity of XML and the consequent time necessary to process it is less of an issue as the message becomes more coarse-grained. While for a simple method invocation the amount of time spent deserializing is a significant amount of the overall time it is relatively small for a document whose contents require complex processing. There are techniques for efficiently extracting information out of large XML documents without having to process the entire document. In this case, sending one large document will usually outperform sending a number of smaller messages using a "chatty protocol".

XML Schema can be thought of as a kind of document strong typing construct. But being adapted to the needs of human beings for representing information structures, it is vastly more flexible than a method signature. A document-centric Web Service loosely couples the services producer and consumer through the XML schema. XML Schema also provides the extension mechanism as a limited way to allow schemas to be modified while maintaining backward type compatibility and target namespaces to allow for distinction between different versions of a schema. In some cases, incompatible documents may be made compatible through transformation services built into the Web Services infrastructure. Document-centric Web Services can therefore be upgraded in certain ways by producers without breaking the consumers's implementations and may be engineered to support version upgrades without affecting the core implementation.

Large-grained document-centric Web Services message exchanges lend themselves naturally to asynchronous communication protocols. Unlike fine-grained operations, the operations which immediately

follow the sending of a request are rarely dependant on the results of that request. Document-centric Web Services will usually have a mechanism for correlating responses and requests through a message identification scheme, allowing for independent, asynchronous processing of messages.

DTDs have been used for interoperation between systems for a long time - XML Schemas for less time but still have accumulated a substantial body of experience. With Schemas the vocabulary of interoperation is unlimited while RPC can only allow expressiveness as far as the simple type system of the programming language will allow.

Document-centric Web Services boils down to doing what XML practioners already know how to do very well - defining complex, semantic-driven applications based on schema-described XML documents. Document-centric Web Services leverages twenty years of collective, chronological experience with structured documents, plus as many years with message-oriented architectures, but puts that experience directly onto the Internet at the service of commerce and communication in the 21st century.

## **Secrets to Implementing High-Performance Document-Centric Web Services**

### **Use a Manifest**

The concept of a manifest is crucial to a successful implementation of document-centric SOAP and to ensuring Web Services interoperability. The manifest is a collection of references that identifies the different payloads of a SOAP Message and includes information about each payload. The manifest allows metadata to be attached to payloads in a transport-binding independent way. It ensures referential integrity during message's transmission and transformation. And it also enables receivers to consume SOAP messages more efficiently.

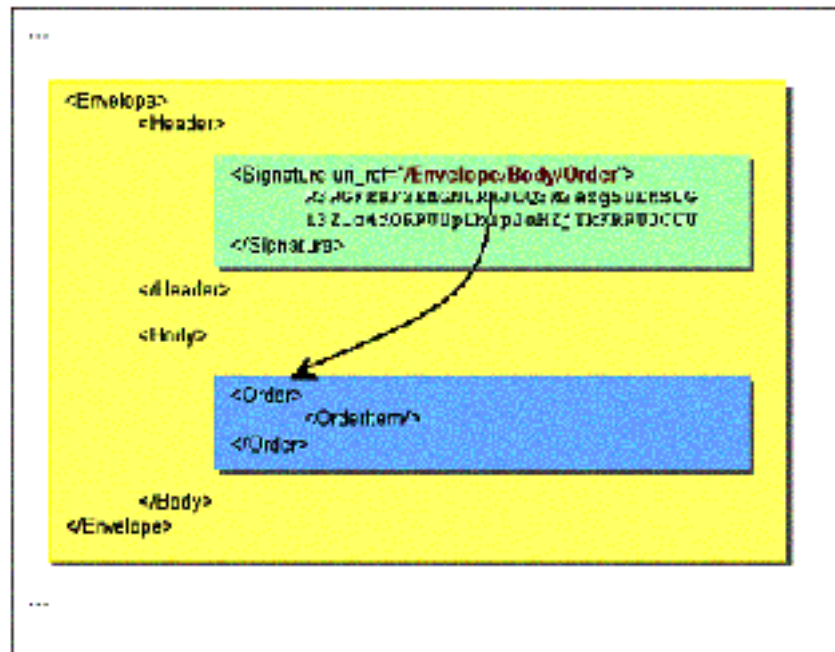
As we know, WSDL serves as "service contract" between Web Service providers and consumers, a key in achieving interoperability in Web Services. To successfully invoke services, service consumers must compose SOAP requests in the format defined in WSDL. The service consumer also knows that the response from the service provider will be constructed according the contract implicit in the WSDL description. The WSDL contract defines the logical content of a service request or response message in the Message Parts section. Each component of the message or message part is identified and labeled by a Part Name. Message receivers use this part name in their logic to determine the content of the incoming SOAP messages and to consume it correctly.

When SOAP was a merely way to serialize remote procedure calls, the simple SOAP message bears nothing but an XMLized method invocation where part names were treated as the names of method parameters and encoded as sub-elements of the SOAP body. In document-centric SOAP, however, there is no defined mapping between part names defined by WSDL and payloads carried in the message. In this case, payloads are treated as opaque documents, being XML or binary, contained within the SOAP body or outside the envelope as attachments. There is no robust algorithm for computing part names from the payload documents themselves.

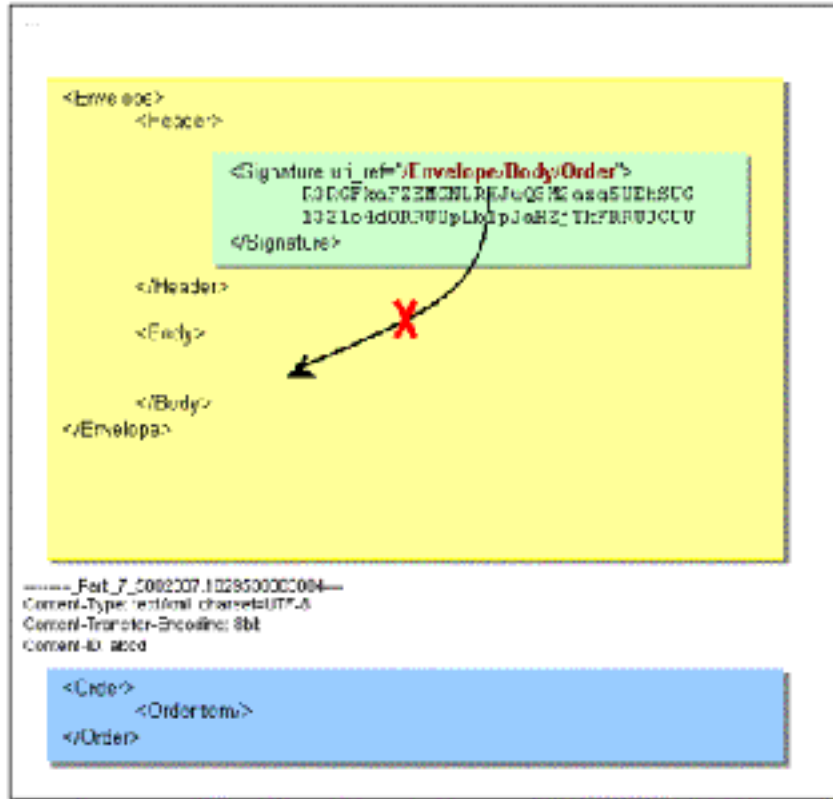
Several proposed solutions, such as using the order by which payloads are located in the message or correlating the root element names to part names, fail hopelessly when the sequence of payload is rearranged as the message passes through intermediate SOAP nodes, or when there are multiple payloads bearing the same root name. Also, the "root-name" idea can't be applied to non-XML payloads at all.

It is the manifest that provides this critical and missing link. Message senders can include part names as part of the payload description in manifest references. The part names captured by the manifest is invariant with respect to payload type (XML or non-XML), document location (body payload or attachment) and how the message is transported. Other payload properties, such as content type and DTD for XML documents, that are important for message processing but cannot be stored in the content of payload itself, can be included in the manifest as well.

Another central function that the manifest can provide is to help maintain the referential integrity of a SOAP message. An intra-message reference is when, by some mechanism, a payload element refers to another payload element located within the same SOAP message. An illustration of this is shown in the following diagram where a URI-Reference in the SOAP signature header references an element located in the SOAP body:



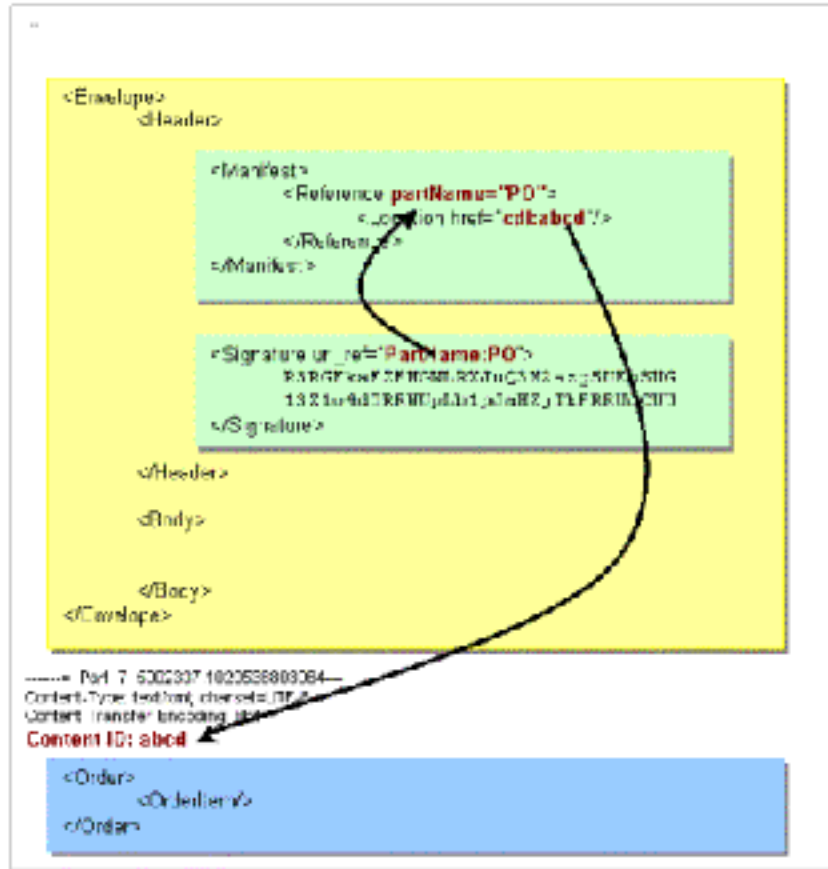
Intra-message referencing is widely used by important message-level services such as WS-Security. It easily breaks down when the SOAP message pass though various intermediate SOAP nodes on its way to the final receiver and has its payload location "adjusted" because of differences in message handling abilities or policies at various nodes. As shown in the following figure, when the order payload is "adjusted" from the SOAP body to an attachment, the original reference becomes invalid, i.e. the referential integrity is broken. Since payloads are treated as opaque documents and the referencing construct is application-specific, there is little hope that the general SOAP framework can "adjust" the reference accordingly.



When the concept of a manifest is introduced into SOAP processing the intra-message referencing problem is surprisingly easy to solve. We begin by proposing a new URI scheme:

**PartName:** <part\_name>

Where "PartName" is the scheme name and <part\_name> is the part name assigned to the payload. With this new URI scheme, a URI reference can be made to a manifest payload entry and the entry itself documents where the payload is actually located, as illustrated by the following figure:



The PartName URI reference is independent to the actual location of the payload. Since the manifest is part of the SOAP framework, the payload location attribute gets adjusted automatically when the payload is moved around or a different transport binding is applied. This indirect referencing mechanism provided by the manifest ensures the message's referential integrity.

Lastly, but not the least important, the SOAP manifest acts much like a table of contents for a book. With the table of contents, you don't need to read the whole book to know roughly what it contains and you can quickly jump to the chapter or section you are interested by following the page number given by the table. The manifest allows message-consuming applications to quickly determine what payloads or parts are contained in the SOAP message and to extract the part they are interested in by following the payload location path without scanning through the whole message. Combined with stream-based processing techniques, the manifest helps to greatly improve the performance of document-centric SOAP handling.

## Use Document-centric Message Processing Techniques

In document-centric SOAP, the task of the SOAP framework is no longer to encode and decode between procedure calls and the XML infoset. The task of the SOAP framework is to carry application-specific data, whether XML or non-XML, defined by XML schema or other typing systems, from sender to receiver. This shift significantly broadens our choices on how the SOAP envelope and payloads can be represented and processed.

It is no longer necessary to represent the entire SOAP envelope as a DOM tree. The tree representation is only important for RPC-style SOAP when the framework has to traverse up and down the tree to interpret the SOAP message into a method invocation. In document-centric SOAP, the "transparency" of SOAP message structure stops at the payload level. SOAP payloads are treated as "opaque" documents - it is neither possible nor necessary for the SOAP framework to figure out the document content, format, the typing systems used or to perform validation beyond very basic requirements of the SOAP specification. It is entirely up to the payload constructor or consumer to figure all these out. In computer science, laziness is esteemed as the greatest of virtues and the preferred processing strategy of all virtuous software designers is called "lazy evaluation". The virtuous document-centric SOAP framework designer will always postpone hard computing work until the moment when it is absolutely necessary.

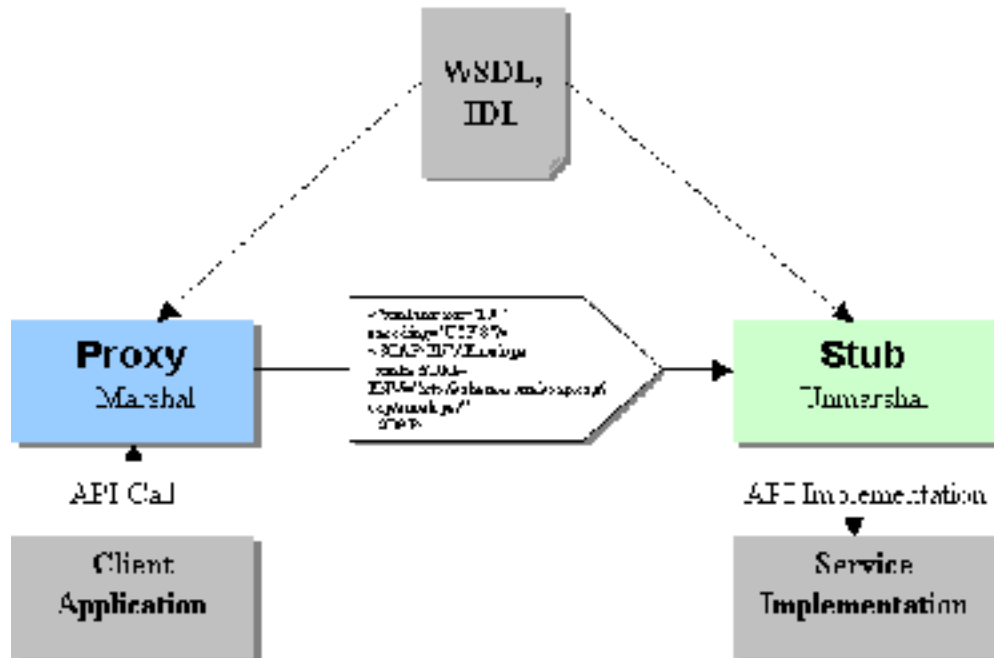
XML programmers have an extremely effective technique for implementing lazy algorithms - stream-based parsing. Using event-based programming APIs such as Simple API for XML (SAX), we can avoid creating an expensive tree structure in memory. All payload documents are kept as byte streams, in the same format they are received from the wire (i.e., read from the transport used to carry the message on the Internet). No unnecessary up-front processing is done to them; lazy SOAP processing will only validate that basic structure of the envelope is correct and the location of payload documents in the SOAP envelope or as attachments conforms to the binding specification. The payload consuming application assumes responsibility for converting the document from a byte stream to any other desired representation, or to perform validation or any other required processing. There are many applications such as archiving which will never need to perform expensive conversions. Other applications may be able to take advantage of new, high-efficiency tools that operate directly on streams. For example, many queries may be performed with sequential or stream-based XPath implementations and most transformations can be performed using streaming transformation engines like STX or Commerce One's XST. By giving the application both the responsibility and the freedom to choose the optimal representation of XML data the cost (memory and CPU) of processing a particular type of message can be made as low as possible and the overall load on the system from handling document-centric SOAP will average much lower.

This discussion points to the need for a uniform document framework, representing opaque payload documents, to be tightly integrated into the document-centric SOAP framework. The application developer should be able to deal in a uniform way with documents whether they are in XML or other non-XML formats, and whether they are located inside the SOAP envelope body or transported as an attachment. For example, there are differences between an XML document stored as an attachment and an XML document stored in the body of the SOAP message. In the latter case, the document is within the context of the SOAP envelope and may not have its own prolog and may use namespace declared in the higher context. In order to provide the application programmer with a uniform view of the the document in either case the SOAP framework must ensure that namespaces and attributes from SOAP envelope and body elements propagate down to the nested document when it is extracted. This process of modifying the document in the SOAP body is "justified" by the notion of canonical equivalency. Canonical equivalency proves to be essential in implementing WS-Security services such as XML signature.

## **Use Generated APIs for Processing Payload Documents**

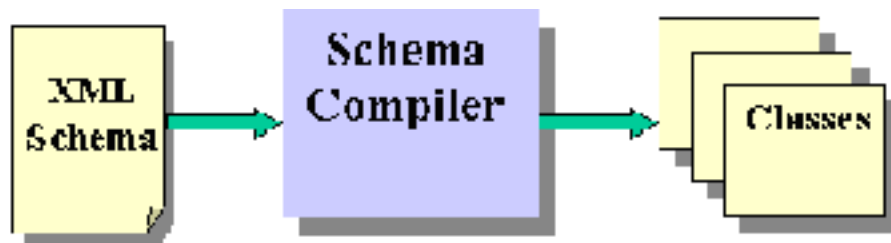
Strong API support is THE reason RPC-Style SOAP gained its popularity quickly among programmers. As illustrated by the following figure, in a typical RPC-Style SOAP development environment, the WSDL file is compiled into proxy and stub by a SOAP development tool. This role of WSDL in this process is very similar to that of Interface Definition Language (IDL) in traditional RPC frameworks such as RMI or CORBA. The proxy and stub are the marshaller and an unmarshaller, respectively, that converts functional calls to and from serialized XML format. The programmer at the service provider's side only needs to use the stub as a template to provide the function implementation and the programmer at the client side only

needs to make function calls to invoke the services. Neither of them needs to be concerned with how functional parameters get correctly encoded into the SOAP envelope.



The lack of appropriate API support might be THE reason why programmers are slow to shift to the document-centric SOAP paradigm. In document-centric SOAP, the majority of message payloads are large XML instances defined by complex XML schemas. These payloads are treated as opaque documents and application programmers have to ensure that they are constructed or consumed correctly. The standard generic APIs for manipulating XML documents such as DOM, JDOM or SAX all require advanced knowledge of the XML infoset and parsing techniques. The programmer is already hard pressed to implement their business application. Forced also to ensure that the SOAP payload is properly constructed, many developers naturally find document-centric SOAP on the whole thing a rather daunting undertaking.

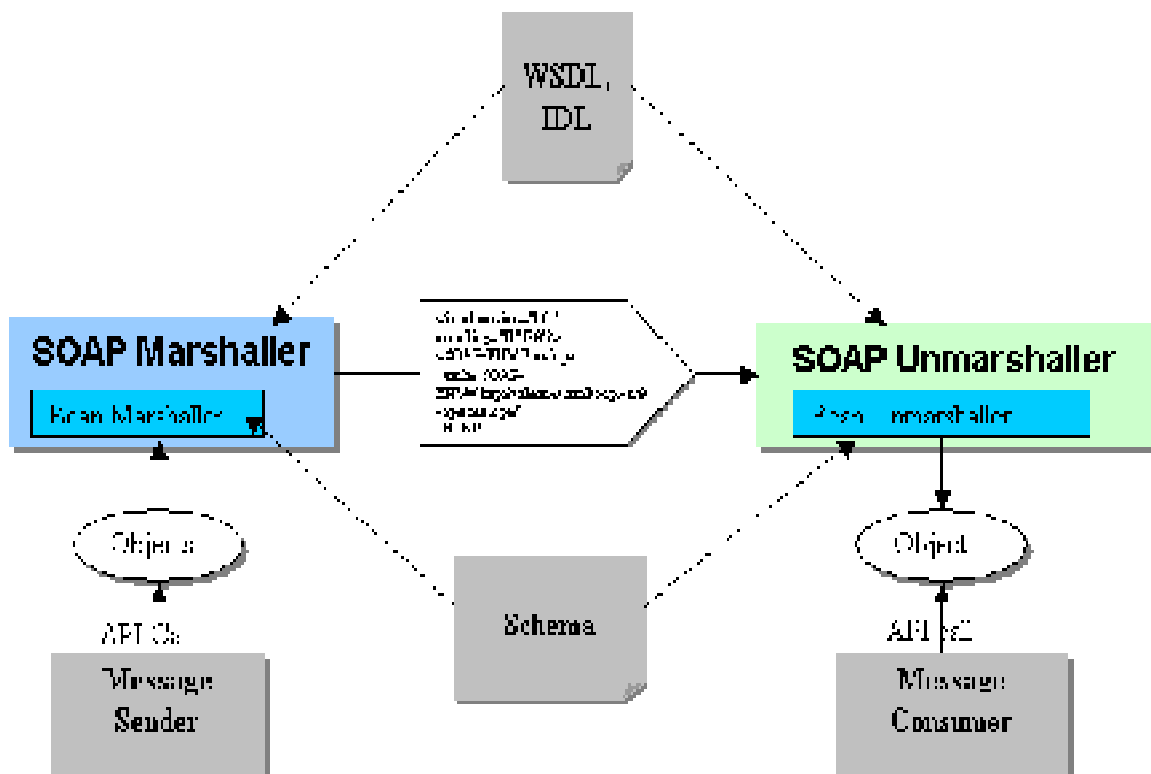
A group of emerging xml-object mapping tools, such as JAXB, Breeze XML Binder and Commerce One's XGen, offer a solution. These tools compile XML schemas into classes in a programming language such as Java.



Different than generic XML programming models, schema-generated classes capture the specific data structures defined by schemas. Since the API is based on the application's schemas, the programmer can be sure that businesses-specific data will be accessed correctly at compile-time (static checking). The programmer does not need any XML knowledge. At run-time, the run-time environment provided by the mapping tool functions as the marshaller/unmarshaller that converts XML instances to/from programming objects automatically while performing schema validation (dynamic checking).



The following figure illustrates a Web Services environment where the schema-compiled object marshaller and unmarshaller are embedded in the SOAP message marshaller and unmarshaller. To construct or consume SOAP payloads, the programmer need only be concerned with the business-specific APIs implemented by the schema-mapped objects. Thus, the xml-object mapping tool enables the programmer in a document-centric SOAP environment, like the RPC-SOAP programmer, to solve the application problem in the way he or she knows best - through APIs in the preferred programming language.



## An Open-Source Document-Centric Web Services Toolkit: DocSOAP XDK

The authors led the architecture and implementation of a Web Services, SOAP and XML toolkit designed to support the Web Service's design goals enunciated in the previous section. The DocSOAP XDK is an open-source Java API, available at <http://developer.commerceone.com/>, which is used as the core infrastructure of Commerce One's Web Services platform Conductor.



5. **High Interoperability.** The DocSOAP Framework supports all SOAP versions (1.1 and 1.2) and wire-level bindings, SOAP with Attachments (MIME) and DIME, which enable you to interoperate with all major Web Services platforms including .NET.
6. **High Extensibility.** The Conductor DocSOAP XDK adopts an open framework design. You can bring in other Web Services features, document formats or XML handling according to your needs with great ease.

## The DocSOAP Framework Message Abstraction Layer

SOAP message components are accessed through handlers that allow for efficient internal handling and guaranteed correctness-by-construction. The message abstraction is independent of most details of physical message formation, allowing the maximum interoperability between SOAP formats (1.1 and 1.2 and packaging schemes and transports). This abstraction allows these details to be resolved at the last possible moment at message marshalling. Complex details of message fragment handling are likewise transparent to the user through the message abstraction.

The user of DocSOAP has the ability to manipulate, not directly but through API calls, the attributes on the Envelope elements qualified by the SOAP schema namespace (Envelope, Header, Body), to create, modify and remove header blocks, and insert and remove payloads from the body. The internal content of the header block is not manipulated through the DocSOAP API; the user may use the Document Framework API for this purpose. At marshalling time the user also has the ability to set the format to SOAP 1.1 or SOAP 1.2. While the user has to understand the purpose of these physical aspects of the SOAP envelope to use the abstraction properly, he or she is not able to corrupt the basic envelope structure by any sequence of API calls.

## Implementation of the Manifest in DocSOAP

### The Manifest Abstraction

The manifest concept in DocSOAP is implemented in two different ways: through an actual SOAP manifest header which can be included in a SOAP message and through an API used by the SOAP programmer. Although the two aspects of the manifest paradigm are complementary either one could, in principle, exist without the other. The manifest SOAP header can be included in the SOAP message but would be accessed like any other header if a special API for it were not provided. The application would find it advantageous to have information about the contents of the message collected in one place but, without an API mediating access to the message parts and to the manifest, it would do nothing to ensure the integrity of the message or to enable an efficient underlying implementation of message handling. If the manifest API were implemented but there were no manifest header the integrity of the message would be guaranteed and the SOAP programmer would enjoy having the use of convenient abstraction for manipulating SOAP messages but there would no way to share the contents of the manifest between SOAP nodes in a processing pipeline

Actually, this is a commonplace situation as DocSOAP is the only tool capable of producing a manifest header and therefore DocSOAP must be able to handle SOAP messages without a manifest coming from

nodes constructed with other SOAP toolkits. When a SOAP message without a manifest is unmarshalled DocSOAP will materialize a manifest based on its analysis of the contents of the message. Certain information, however, cannot be materialized and this may impose additional constraints or work on the SOAP application. For example, the manifest header can contain the PARTNAME supplied through the WSDL service description (for document style SOAP, unlike RPC, this has no direct mapping to the SOAP message). The PARTNAME tells the SOAP application which message part is which. In the absence of a manifest the SOAP application will have to derive the PARTNAME by some out-of-band mechanism and each SOAP node in the pipeline will have to repeat this process.

A very reasonable question to ask about the manifest header is whether the SOAP application can "trust" its contents. Generally - no! There are circumstances under which some level of trust is possible. It may be known that a "trustworthy" API like DocSOAP that is explicitly designed to guarantee both the integrity of the SOAP message and the synchronization between the manifest header and the SOAP message was used to construct the SOAP message with its manifest. The manifest will be even trustworthier if the SOAP message and the manifest header are digitally signed. But in general, the manifest header should be verified on unmarshalling. DocSOAP does this although validation may be "lazy", e.g., an entry for a message page may not be validated until some operation is actually performed on that part by the SOAP application. However, the manifest serialized out by DocSOAP on marshalling will always be 100% valid.

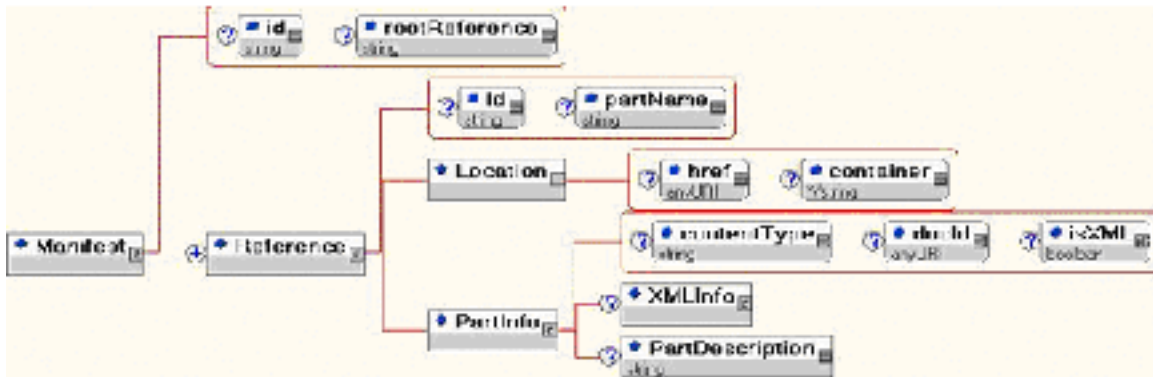
It is clear from the preceding discussion that while the manifest header and the manifest API do not have to be combined, the combination of the two is the very best means for ensuring SOAP message integrity, enabling an efficient implementation, and providing the most convenience and functionality to the SOAP programmer.

## **Manifest Wire Format**

Commerce One has created a SOAP manifest header defined by an XML schema and supported by DocSOAP.

The manifest is composed of a set of References. Each Reference contains metadata for one message part including: location identifier, partname, media type, and XML processing information. This information is often essential for receiving applications to process SOAP document parts but cannot be included in the payload itself. The manifest as a SOAP extension header block provides a standard and packaging independent way of communicating payload properties from senders to receivers.

A graphic view of the Commerce One manifest schema is shown below.



## The Manifest API

The key thing to understand about the manifest programming paradigm is that all operations on message parts (add, remove, inspect, set properties, etc.) are mediated through the manifest API (manifest and descriptor classes). It is not possible to manipulate the SOAP message in any other way. This approach guarantees synchronization between the manifest (either in memory or when marshalled into XML) and the SOAP message it describes. Descriptors are similar to "handles"; in order to obtain a SOAP message part you first obtain its descriptor. The location field of the descriptor is a special, protected field; although the value of this field can be set by the user in order to accommodate certain special requirements, the user is not encouraged to do so since the API is responsible for determining valid values for this pointer into the SOAP message and will always set this field correctly automatically. In any case, DocSOAP will always reject impermissible location pointers that the user attempts to set.

The manifest API simplifies the task of programming SOAP. The user of DocSOAP does not need to know or manage any of the details of SOAP packaging. This level of abstraction also makes it possible for DocSOAP to support all current packaging formats and versions of SOAP as the details of writing out the SOAP message can be postponed until marshalling time. The association of metadata with SOAP message parts gives the application developer many possible ways to query for parts including the SOAP role or actor, WSDL PARTNAME, designation as the root part, document identifier and so on.

The one special case is the SOAP Fault. Although the SOAP Fault is carried in the SOAP payload it is not described in manifest, although the manifest indicates the existence of a fault through the hasFault property. Access to SOAP faults is mediated through another part of the Doc SOAP API, the SOAP Fault classes.

Although the manifest is a SOAP header the manifest cannot be manipulated through the SOAP header classes. The manifest header can be either marshalled or not marshalled into XML depending on a setting on the writeto method.

## The Document Framework

The Document Framework gives programmers a single class library for handling all types of documents: binary, text and XML. The Document Framework class library implements a single-rooted deep hierarchy of document types that provides the ideal level of abstraction for each of a set of operations on documents.

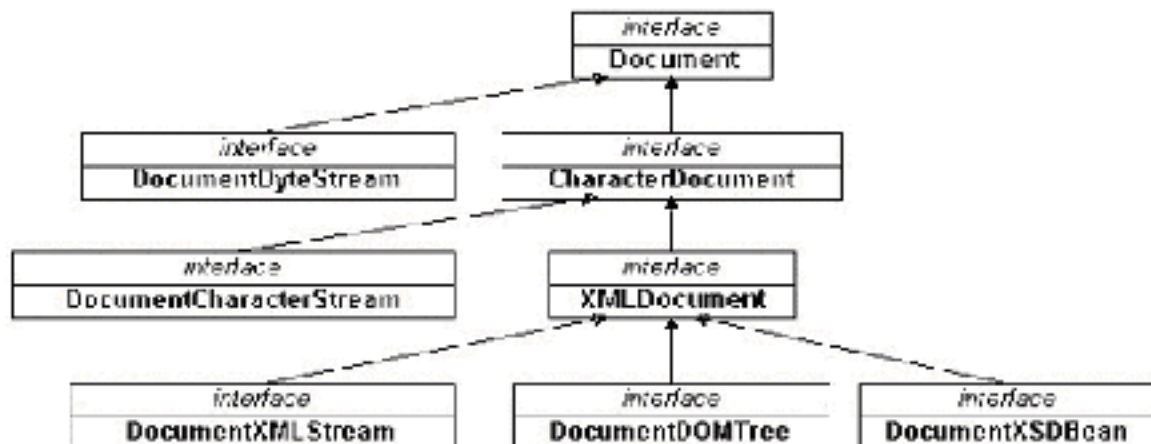
The Document Framework unites and interrelates all major programming interfaces to XML documents: SAX, DOM, and Schema-generated Java Beans.

The Document Framework is designed for high performance, especially in the handling of larger XML documents. The overall principle used to enable high performance is to provide a choice of lower to higher cost operations and to enable the user to avoid high cost operations unless or until they are necessary. The hierarchy of document types provides the range of low to higher cost representation and the possibility of up (and down) conversion. The user can often avoid converting documents to very costly representations such as a DOM tree or Java bean. The Java Activation Framework's Data Source is used to further the performance strategy of lazy processing by keeping the document in its source location until the document data is actually needed. The Document Framework uses low-memory, efficient stream processing wherever possible.

The Document Framework defines document type classes as interfaces. Additional implementations which fulfill the interface may be added or replace existing implementations.

Documents in the Document Framework are self-contained, self-descriptive objects carrying a URL identifier, a name, and type information. A document framework document is uniform irrespective of creation method or original storage medium.

The relationship between the different document representations used in the Document Framework is illustrated in the following class hierarchy diagram:

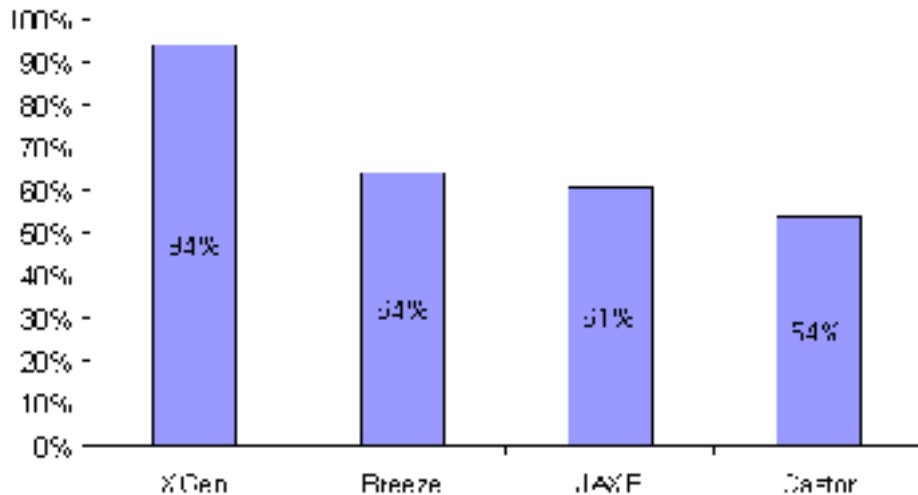


## XGen - API Generator for XML Schemas

The Commerce One XGen (XML Schema Code Generation) is a tool used to translate XML Schema structures into a Java API, enabling any Java developer to do intelligent XML processing. In the context of Web Services such a tool may replace the stub and skeleton generation performed by on an RPC described in a service's WSDL file.

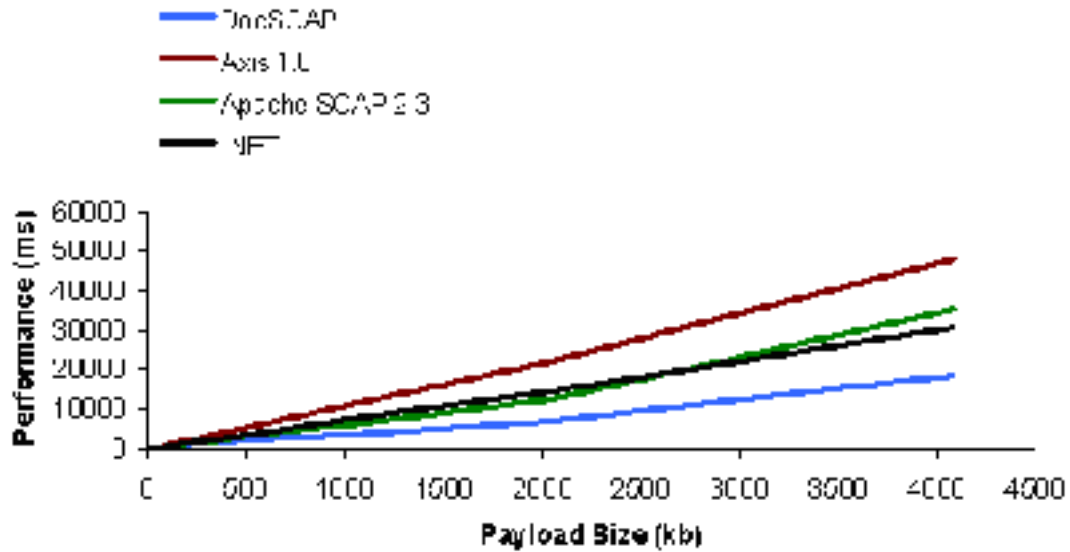
Compared to the generic approaches of DOM and JDOM, XGen provides a solution with a custom fit to the schema structure describing the XML documents. Therefore solutions can be achieved in less time, with less effort spent both in development and testing, with less error-prone results. In addition, the developers do not need to know any XML, but will still produce 100% valid documents.

XGen is similar to tools such as the JAXB reference implementation and Castor. In fact, XGen was from the Castor code base, although over a year of work was spent to rewrite and improve it with the result that the tool now bears little resemblance to its parent. Compared to these other approaches, XGen is the only approach where the mapping between XML Schema and Java code is non-configurable and well-defined, enabling identical code to be generated by anyone at any time and place. This makes the structure of the generated code predictable and reliable, and easily communicable between developers, and decreases errors due to differences in generated code caused by configuration differences or changes. We felt that in a distributed Web Services environment this predictability and reliability was absolutely essential. In addition, XGen is the most complete mapping out of the three, and the only one that does not fail on valid values due to the mapped type not being able to handle the full range of valid values. The following chart shows the percentage of schema features handled correctly by XGen compared to these other tools:



## Performance of the DocSOAP XDK

The DocSOAP XDK is approximately twice as fast as other SOAP toolkits when performing document-centric Web Services.



## What Needs to be Done?

We feel that Web Services will, indeed, prove to one of the next revolutions in Internet computing. The primary importance of Web Services will be that it will make the powerful XML technologies that have been evolving over the last 10 or even 20 years work in a standard, interoperable way over and through the Internet. There is still much to be done to bring this vision to fruition:

- *Understand the new paradigm* - document-centric Web Services. We don't discount the importance of SOAP-RPC for some subset of Web Services applications but the attention and education should shift over to Web Service's "other" paradigm.
- *Provide tools*. We think our DocSOAP XDK is a good start but we'd like to see a range of choices enabling developers to create a wide range of document-centric Web Services applications.
- *Make the Manifest a Web Services standard*. We are putting a specification out. for public discussion. Once the manifest concept is embraced there will most likely be some evolution of our proposal.
- *Make WSDL interoperable for document-centric Web Services*. We've only touched on some of the problems with WSDL in this paper. WSDL provides a complete mapping for RPC-style Web Services, but not for document-centric services. There is work to be done around mapping the operation, partnames, specification of attachments and several other issues. Some of these issues will be solved in conjunction with the adoption of a manifest specification.

- *Address the intra-message reference problem.* Solutions in the digital signature specification and other places are transport-bound and fragile. Through the use of indirect references in the manifest and a standard transformation algorithm for going between local and intra-message references systems the reference problem can be completely solved.
- *Address "XML" problems with namespaces, prefix rewriting, namespace propagation, and document fragments.* Knotty problems all, and much debated in the XML world. These problems can all be solved fairly easily in the Web Services context and the Web Service context makes it critical to do so in the short term.

## Bibliography

[CAIP]

Latency Performance of SOAP Implementations <http://www.caip.rutgers.edu/TASSL/Papers/p2p-p2pws02-soap.pdf>

[ACM]

The Deliberate Revolution Transforming Integration with XML Web Services [http://www.acmqueue.org/issue/turner1.cfm?client\\_no=NEW](http://www.acmqueue.org/issue/turner1.cfm?client_no=NEW)